High-Performance Web Crawling (Marc Najork, Allan Heydon, 2001)

초록  
고성능 웹 크롤러는 많은 웹 서비스의 중요한 구성요소입니다. 예를 들어, 검색 서비스는 웹 크롤러를 사용하여 색인을 채우고, 비교 쇼핑 엔진은 웹 크롤러를 사용하여 온라인 판매업체로부터 제품 및 가격 정보를 수집하고, Internet Archive는 이를 사용하여 인터넷의 역사를 기록합니다. 고성능 크롤러 설계는 주로 웹의 대규모로 인해 기술적 및 사회적 문제를 제기한다. 웹 크롤러는 매우 높은 속도로 페이지를 다운로드할 수 있어야 하지만 특정 웹 서버를 압도해서는 안 됩니다. 또한 메인 메모리에 저장하기엔 너무 큰 데이터 구조를 유지해야 하지만 효율적으로 액세스하고 업데이트할 수 있어야 한다. 이 장에서는 이러한 고성능 크롤러를 제작하고 작동한 경험을 설명합니다.

1 소개  
웹 크롤러(Web Crowler, 웹 로봇 또는 스파이더)는 웹 페이지를 다운로드하는 프로그램이다. 일련의 "seed" Uniform Resource Locators (URLs)가 주어진 경우, 크롤러는 반복적으로 s에서 하나의 URL을 제거하고, 해당 페이지를 다운로드하고, 여기에 포함된 모든 URLs을 추출하고, 이전에 알 수 없었던 URL을 s에 추가합니다. 웹 크롤링 알고리즘은 개념적으로 간단하지만 주요 검색 엔진이 사용하는 것과 유사한 고성능 웹 크롤러를 설계하는 것은 복잡한 작업이다. 이러한 고성능 크롤러를 구축하는 데 내재된 모든 과제는 결국 웹의 확장 때문입니다. 한 달에 10억 페이지를 크롤링하기 위해서, 크롤러는 매초마다 약 400페이지를 다운로드 받아야 한다. 또한, 크롤러는 메인 메모리의 한계를 넘어서는 확장성이 뛰어나야 하는 몇 가지 데이터 구조(예: 다운로드될 나머지 URL 집합)를 저장해야 합니다.  
 우리는 Mercator라는 고성능 웹 크롤러를 구축했는데, 이는 다음과 같은 특성을 가지고 있습니다.

Distributed. Mercator 크롤은 성능을 향상시키기 위해 여러 시스템에 대칭으로 배포할 수 있습니다.

Scalable. Mercator는 두 가지 측면에서 확장 가능하다. 첫째, 분산형 아키텍처로 인해 Mercator의 성능은 크롤링 클러스터에 추가 시스템을 추가하여 확장할 수 있습니다. 둘째, Mercator는 빠르게 성장하는 웹에 대처할 수 있도록 설계되었다. 특히, 그것의 데이터 구조는 탐색 중인 웹의 크기에 관계없이 제한된 양의 메인 메모리를 사용한다.  
이는 대부분의 데이터를 디스크에 저장함으로써 달성됩니다.

High performance. 4대의 Compaq DS20E 666 MHz 알파 서버에서 실행되고 160 Mbit/sec 인터넷 연결을 포화상태로 만든 가장 최근의 크롤링 동안 Mercator는 17일 동안 하루에 약 5천만 개의 문서를 다운로드했습니다.

Polite. 속도가 필요함에도 불구하고, 웹 서버에 과부하를 주는 웹 크롤러를 실행하는 사람은 곧 그러한 행동이 용납될 수 없는 것으로 간주된다는 것을 알게 된다. 적어도 웹 크롤러는 동일한 웹 서버에서 여러 페이지를 동시에 다운로드하려고 해서는 안 되며, 웹 서버가 사용하는 리소스 부분에 제한을 두어야 합니다. Mercator은 이러한 politeness policies 중 어느 하나에 따르도록 설득될 수 있다.

Continuous. 이전에 다운로드한 페이지를 지속적으로 refetch 하는 것이 바람직한 많은 응용 프로그램(새 검색 엔진 인덱스 유지 등)이 있습니다. 이는 자연스럽게 오래된 페이지의 다운로드와 새로 발견된 페이지의 다운로드를 어떻게 상호 분리시킬 것인가 하는 문제를 제기한다. Mercator는 URL 다운로드를 예약하기 위한 우선순위 기반 메커니즘을 제공함으로써 이 문제를 해결한다.

Extensible. 두 크롤링 업무는 같지 않다. 이상적으로 크롤러는 제3자가 새로운 기능을 추가할 수 있는 모듈 방식으로 설계되어야 한다. Mercator는 구성요소 기반 아키텍처를 통해 이러한 이상을 달성한다. Mercator의 각 주요 구성 요소는 추상 인터페이스에 의해 지정된다. 우리는 각 구성요소의 수많은 구현을 작성했으며, 제3자는 처음부터 새로운 구현을 작성하거나 객체 지향 하위 분류를 통해 우리의 구현을 확장할 수 있다. 특정 크롤링 작업에 대해 Mercator를 구성하기 위해 사용자는 적절한 구성 요소가 동적으로 로드되도록 하는 구성 파일을 제공한다.  
  
Portable. Mercator는 전적으로 Java로 작성되므로 Java 가상 머신이 있는 모든 플랫폼에서 실행됩니다. 특히 윈도 NT, 리눅스, 트루64 유닉스, 솔라리스, AIX에서 구동되는 것으로 알려져 있다.

hight perfomance 요구사항과 scalability, politeness, extensibility 및 portability 요구사항 사이에는 자연스러운 긴장감이 있다. 이러한 모든 기능을 동시에 지원하는 것은 중요한 설계 및 엔지니어링 과제이다.  
이 장에서는 Mercator의 설계와 구현, 그것을 만드는 과정에서 배운 교훈, 그리고 대형 크롤을 수행하는 우리의 경험에 대해 설명합니다.

2 웹 크롤러 조사

웹 크롤러는 웹 자체만큼이나 오래되었습니다 [16]. 최초의 크롤러 매튜 그레이의 방랑자(Matthew Gray's Wandererer)는 1993년 봄에 쓰여졌으며, NCSA 모자이크의 첫 번째 발매와 거의 일치했다[9]. 웹 크롤링에 관한 몇몇 논문은 처음 두 번의 월드 와이드 웹 컨퍼런스에서 제시되었다 [7, 18, 20]. 그러나 당시에는 웹이 현재보다 3~4배 작았기 때문에 이러한 시스템은 오늘날의 웹에 내재된 확장 문제를 해결하지 못했습니다.  
  
분명히, 인기 있는 모든 검색 엔진은 웹의 상당 부분까지 확장되어야 하는 크롤러를 사용한다. 그러나, 검색 엔진 사업의 경쟁적인 특성 때문에, 이러한 크롤러들의 디자인은 공개적으로 설명되지 않았다. 구글 크롤러(Google Crowler)와 인터넷 아카이브 크롤러(Internet Archive Crowler) 두 가지 예외가 있다. 불행히도, 문헌에서 이러한 크롤러에 대한 설명은 너무 간결해서 재현을 할 수 없다.

오리지널 구글 크롤러[2] (Stanford에서 개발)는 서로 다른 프로세스에서 실행되는 5개의 기능 컴포넌트로 구성되었다. URL 서버 프로세스가 파일에서 URL을 읽고 이를 여러 크롤러 프로세스로 전달합니다. 각 크롤러 프로세스는 서로 다른 시스템에서 실행되었고, 단일 스레드로 처리되었으며, 비동기 I/O를 사용하여 최대 300대의 웹 서버에서 병렬로 데이터를 가져옵니다. 크롤러는 다운로드된 페이지를 단일 StoreServer 프로세스로 전송하여 페이지를 압축하여 디스크에 저장했습니다. 그런 다음 인덱서 프로세스에 의해 페이지를 디스크에서 다시 읽었으며, 이 프로세스는 HTML 페이지에서 링크를 추출하여 다른 디스크 파일에 저장했다. URL resolver 프로세스는 링크 파일을 읽고, 여기에 포함된 URL의 상관 관계를 제거한 다음, URL 서버가 읽은 디스크 파일에 절대 URL을 저장했습니다. 일반적으로 3~4대의 크롤러 머신이 사용되었기 때문에 전체 시스템은 4~8대의 머신이 필요했다.

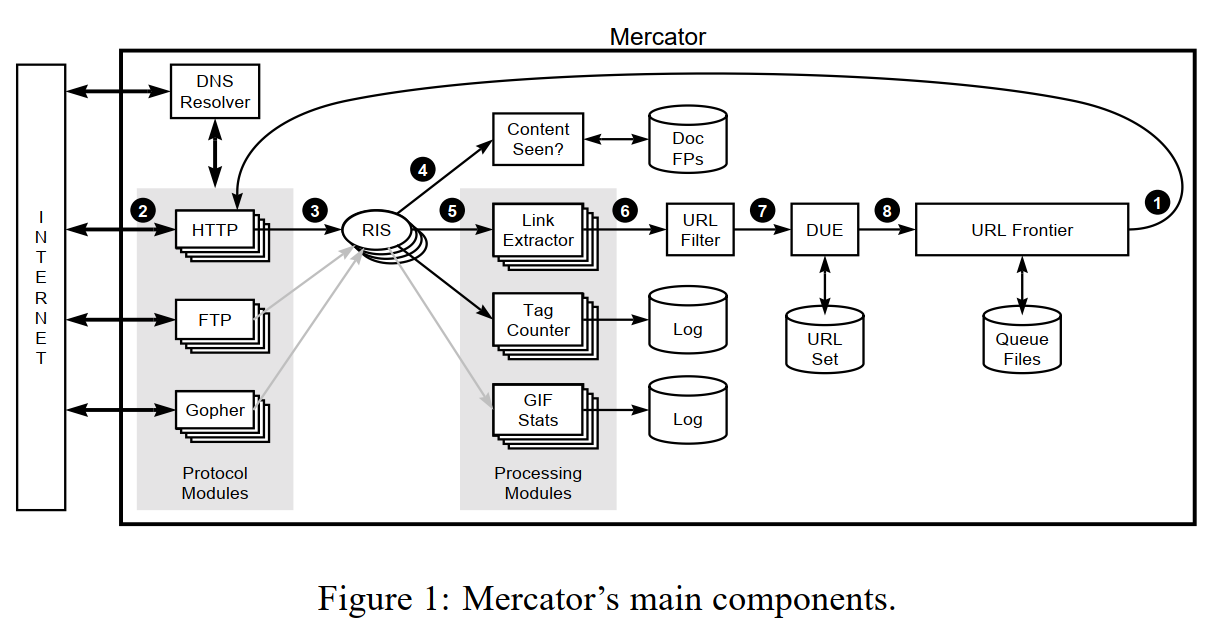
웹 크롤링에 대한 연구는 구글이 상업적인 노력으로 변모한 후에도 스탠포드 대학에서 계속되고 있다. Stanford WebBase 프로젝트는 초당 50개에서 100개의 문서를 다운로드할 수 있는 고성능 분산 크롤러를 구현했습니다 [14]. Cho와 다른 사람들은 또한 incremental crawlers 의 다운로드 일정을 알리기 위해 문서 업데이트 주파수 모델을 개발했다 [5].

또한 Internet Archive는 여러 대의 컴퓨터를 사용하여 웹을 탐색했습니다 [4, 15]. 각 크롤러 프로세스에는 최대 64개의 사이트가 할당되었으며 둘 이상의 크롤러에는 사이트가 할당되지 않았습니다. 각 단일 스레드 크롤러 프로세스는 할당된 사이트에 대한 시드 URL 목록을 디스크에서 사이트별 대기열로 읽은 다음 비동기 I/O를 사용하여 이러한 대기열에서 페이지를 병렬로 가져옵니다. 페이지가 다운로드되면, 크롤러는 페이지에 포함된 링크를 추출했습니다. 링크가 포함된 페이지의 사이트를 참조하는 경우 해당 사이트 큐에 추가되고, 그렇지 않으면 디스크에 기록됩니다. 주기적으로, 배치 프로세스는 기록된 "사이트 간" URL을 사이트별 시드 세트에 병합하여 프로세스의 중복을 필터링하였다.

WebFountain 크롤러는 Mercator의 몇 가지 특성을 공유한다: 그것은 distributed, continuous (저자들은 “incremental”이라는 용어를 사용함), polite, and conﬁgurable [6]이다. 불행하게도, 이 글의 현재, WebFountain 는 개발 초기 단계이며, 그것의 성능에 대한 데이터는 아직 이용할 수 없다.

3. Mercator’s Architecture

웹 크롤러가 실행하는 기본 알고리즘은 시드 URL 목록을 입력으로 가져와서 다음 단계를 반복적으로 실행합니다. URL 목록에서 URL을 제거하고 호스트 이름의 IP 주소를 확인한 후 해당 문서를 다운로드하고 URL에 포함된 링크를 추출합니다. 추출된 각 링크에 대해 해당 링크가 절대 URL인지 확인하고(필요한 경우 삭제) 다운로드할 URL 목록에 추가합니다(이전에는 발견되지 않았음). 원하는 경우, 다운로드된 문서를 다른 방식으로 처리합니다(예: 문서 내용을 색인화).



이 기본 알고리즘에는 여러 가지 기능 구성 요소가 필요합니다.  
• 다운로드할 URL 목록을 저장하는 구성요소(URL 프론티어라고 함)  
• 호스트 이름을 IP 주소로 확인하는 구성 요소  
• HTTP 프로토콜을 사용하여 문서를 다운로드하기 위한 구성요소  
• HTML 문서에서 링크를 추출하기 위한 구성요소  
• URL이 이전에 발견되었는지 여부를 결정하는 구성 요소.

이 섹션의 나머지 부분에서는 메르카토르(Mercator)가 이 기본 알고리즘을 어떻게 재구성하는지 설명한다.  
 그림 1은 메르카토르(Mercator)의 주요 구성 요소를 보여줍니다. 크롤링은 일반적으로 수백 개의 숫자를 가진 여러 작업자 스레드에 의해 수행됩니다. 각 작업자는 문서를 다운로드하고 처리하는 데 필요한 단계를 반복적으로 수행합니다. 이 루프 ①의 첫 번째 단계는 다운로드를 위해 공유 URL 프런티어로부터 절대 URL을 제거하는 것이다.  
 절대 URL은 이를 다운로드하기 위해 사용되어야 하는 네트워크 프로토콜을 식별하는 체계(예: "http")로 시작한다. Mercator에서 이러한 네트워크 프로토콜은 프로토콜 모듈에 의해 구현된다. 크롤에서 사용할 프로토콜 모듈은 사용자 제공 구성 파일에 지정되며 크롤 시작 시 동적으로 로드된다. 기본 구성은 HTTP, FTP, Gopher를 위한 프로토콜 모듈을 포함한다.  
 URL의 체계에 따라 작업자는 문서를 다운로드하기 위한 적절한 프로토콜 모듈을 선택합니다. 그런 다음 프로토콜 모듈의 가져오기 방법을 호출하여 인터넷 ②에서 스레드당 문서를 다운로드합니다.

RewindInputStream③ (또는 줄여서 RIS) RIS는 임의의 입력 스트림에서 초기화된 I/O 추상화이며, 이후에 해당 스트림의 콘텐츠를 여러 번 다시 읽을 수 있게 한다.  
정중한 웹 크롤러는 웹 마스터가 사이트 일부를 크롤러에 대해 제한되지 않음을 선언할 수 있는 로봇 제외 프로토콜을 구현합니다 [17]. 로봇 제외 프로토콜은 웹 크롤러가 실제 콘텐츠를 다운로드하기 전에 웹 사이트에서 이러한 선언을 포함하는 "/robots.txt"라는 이름의 리소스를 가져올 것을 요구한다. 모든 요청에서 이 자원을 다운로드하는 것을 피하기 위해, Mercator의 HTTP 프로토콜 모듈은 그들의 로봇 제외 규칙에 호스트 이름을 매핑하는 fixed-size를 유지한다. 기본적으로 캐시는 218개 항목으로 제한되며 LRU 대체 전략을 사용합니다. 문서가 RIS에 작성되면, 작업자 스레드는 동일한 내용이지만 다른 URL을 가진 문서가 ④ 이전에 보였는지를 확인하기 위해 내용 보기 테스트를 실행합니다. 그러면 해당 문서는 더 이상 처리되지 않고 작업자 스레드는 ①단계로 돌아갑니다.

다운로드한 모든 문서에는 내용 유형이 있습니다. Mercator 구성 파일은 스키마를 프로토콜 모듈과 연결하는 것 외에도 하나 이상의 처리 모듈과 콘텐츠 유형을 연결한다. 처리 모듈(processing module)은 HTML 페이지에서 링크를 추출하거나 HTML 페이지에서 발견된 태그를 계산하거나 GIF 이미지에 대한 통계를 수집하는 등 다운로드된 문서를 처리하기 위한 추상화이다. 일반적으로 처리 모듈은 자체 내부 상태뿐만 아니라 크롤러 상태에도 부작용을 일으킬 수 있습니다.

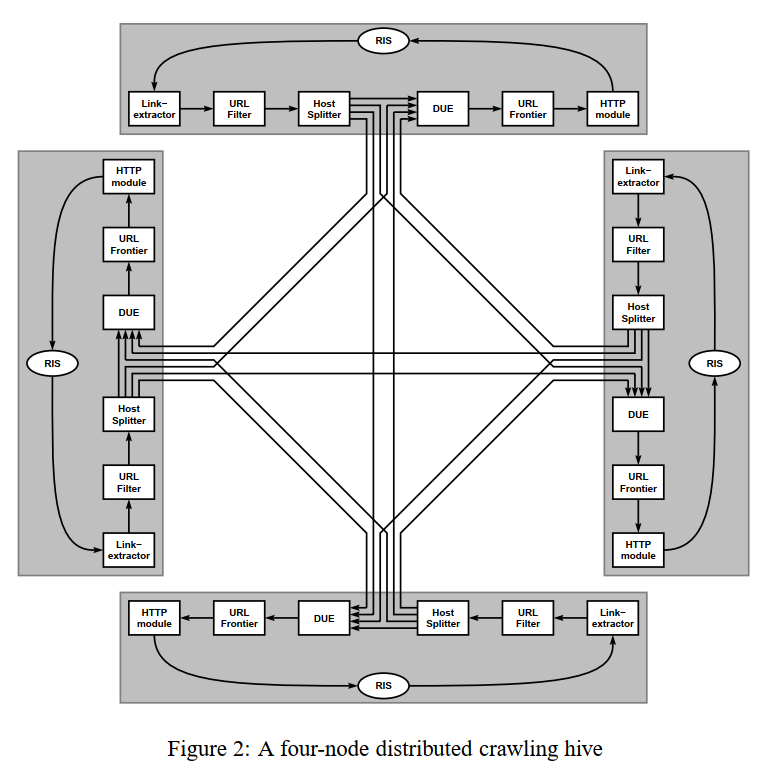
다운로드된 문서의 내용 유형을 기준으로 작업자는 해당 내용 유형 5와 관련된 각 처리 모듈의 처리 방법을 호출합니다. 예를 들어 그림 1의 링크 추출기 및 태그 카운터 처리 모듈은 텍스트/html 문서에 사용되고 GIF Stats 모듈은 이미지/gif 문서에 사용됩니다.

기본적으로 링크 추출을 위한 처리 모듈은 내용 유형 text/html과 연결됩니다. 이 모듈의 프로세스 방법은 HTML 페이지에서 모든 링크를 추출합니다. 각 링크는 절대 URL로 변환되어 사용자가 제공한 URL 필터에 대해 테스트되어 다운로드 여부를 결정합니다 ⑥. URL이 필터를 통과하면 중복 URL 제거기(DUY) (7)에 제출되며, URL이 이전에 확인되었는지 또는 이미 다운로드되었는지 확인합니다. URL이 새 URL이면 Frontier (8)에 추가됩니다.

마지막으로, 지속적인 크롤링의 경우, 방금 다운로드한 문서의 URL도 URL 프론티어에 다시 추가됩니다. 앞에서 언급한 바와 같이, 새로운 URL과 이전 URL의 다운로드를 인터리빙하기 위해서는 연속적인 크롤링 케이스에 메커니즘이 필요하다. Mercator는 이러한 목적을 위해 무작위 우선순위 기반 체계를 사용한다. 연속 크롤링을 위한 표준 구성은 일반적으로 다운로드 내역에 기초하여 URL에 우선 순위를 붙이고, 디큐 방법이 더 높은 우선 순위 URL로 편향되는 프런티어 구현을 사용한다. 바이어스의 정도와 URL 우선순위를 계산하는 알고리즘은 모두 플러그형 구성요소이다. 우리의 구성 중 하나에서, 한 다운로드에서 다음 다운로드로 변경되지 않는 문서의 우선순위는 시간이 지남에 따라 감소하므로 자주 변경되는 문서보다 덜 자주 다운로드됩니다.

문서를 다운로드하고 처리하는 수많은 작업자 스레드 외에도 모든 Mercator 크롤에는 다양한 작업을 수행하는 단일 백그라운드 스레드도 있습니다. 백그라운드 스레드는 주기적으로 웨이크업(기본적으로 10초마다)하고, 크롤 진행에 대한 요약 통계를 기록하고, 크롤이 종료되어야 하는지(프런티어가 비어 있거나 사용자가 지정한 시간 제한을 초과했기 때문) 확인하고, 크롤 상태를 안정적인 스토리지로 확인할 때인지 확인합니다.  
  
 체크포인트 지정은 웹 탐색과 같은 장기 실행 프로세스의 중요한 부분입니다. 체크포인트를 통해 우리는 크롤러 상태를 안정적인 스토리지에 기록한다는 것을 의미하며, 실패 시 크롤러가 체크포인트를 읽음으로써 자신의 상태를 회복하고 체크포인트 당시의 정확한 상태에서 크롤러를 재개하기에 충분하다. 이 정의에 따르면, 실패의 경우, 가장 최근의 체크포인트 이후에 수행된 모든 작업은 손실되지만, 가장 최근의 체크포인트까지의 작업은 없습니다. Mercator에서 백그라운드 스레드가 체크포인트를 수행하는 빈도는 사용자가 구성할 수 있습니다. 일반적으로 하루에 1-4번 정도 체크포인트를 사용합니다.  
  
 지금까지의 설명은 모든 Mercator 스레드가 단일 프로세스에서 실행되는 경우를 가정했다. 그러나 Mercator는 다중 프로세스 분산 시스템으로 구성할 수 있습니다. 이 구성에서 한 공정은 여왕으로 지정되고, 다른 공정은 드론[[1]](#footnote-1)으로, 여왕과 드론은 모두 작업자 스레드를 실행하지만, 여왕만이 통계 기록, 크롤 종료, 체크포인트 개시 등을 담당하는 백그라운드 스레드를 실행한다.

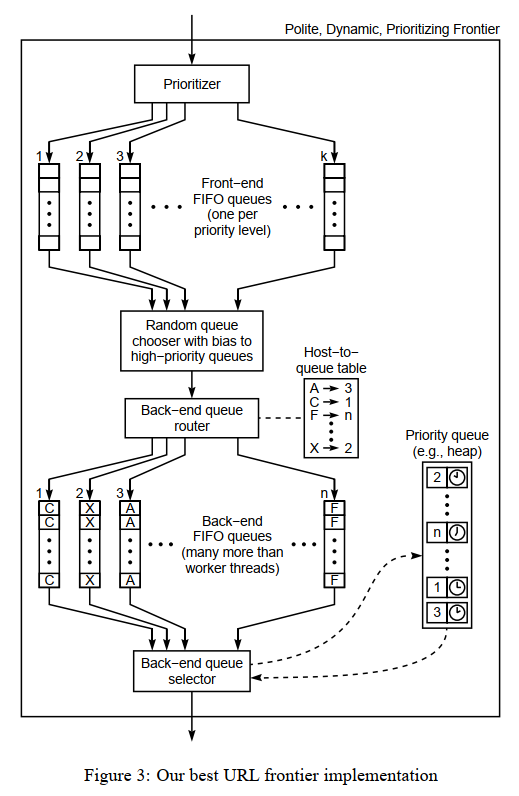
분산형 구성에서 호스트 이름의 공간은 퀸과 드론 프로세스로 분할된다. 각 프로세스는 할당된 호스트 이름의 하위 집합만 담당합니다. 따라서, 각 크롤링 프로세스의 중앙 데이터 구조(URL 프런티어, DUE 에 의해 유지되는 URL 집합, DNS 캐시 등)는 그것의 호스트를 위한 데이터만을 포함한다. 달리 말하면, Mercator 크롤의 상태는 퀸과 드론 프로세스에 걸쳐 완전히 분할됩니다. 데이터의 복제는 없습니다.  
  
 Distributed Crawl에서 링크 추출기가 다운로드된 페이지에서 URL을 추출하면 해당 URL이 URL 필터를 통해 호스트 스플리터 구성 요소로 전달됩니다. 이 구성 요소는 URL의 호스트 이름이 이 프로세스에 할당되었는지 여부를 확인합니다. 이 프로세스에 할당된 프로세스는 DUIL로 전달되고, 다른 프로세스는 적절한 피어 프로세스로 라우팅되며, 여기서 해당 프로세스의 DUIL 구성 요소로 전달됩니다. 링크의 약 80%가 상대적이기 때문에 발견된 URL의 대부분은 이들을 발견한 크롤링 프로세스의 로컬에 남아 있다. 또한 Mercator는 효율성을 위해 아웃바운드 URL을 일괄 전송할 수 있도록 버퍼링합니다. 그림 2는 이러한 설계를 보여줍니다.  
  
 위의 설명에는 몇 가지 중요한 구현 세부 정보가 누락되어 있습니다. 수억 개의 항목을 효율적으로 처리할 수 있는 데이터 구조를 설계하는 것은 많은 엔지니어링 과제를 제기한다. 이러한 우려의 핵심은 메모리 사용과 성능의 균형을 유지해야 한다는 것입니다. 다음 하위 섹션에서는 URL 프런티어, DUTY 및 DNS 확인 프로그램 구성 요소에 대한 추가 세부 정보를 제공합니다. 아키텍처와 구현에 대한 자세한 설명은 다른 곳에서 확인할 수 있습니다 [12, 19].



3.1 예의 바르고 우선 순위를 정하는 URL 프런티어  
  
 모든 웹 크롤러는 다운로드할 URL을 추적해야 합니다. 우리는 이러한 URL을 저장하기 위한 데이터 구조를 URL 프런티어라고 부른다. 이름에도 불구하고 Mercator의 URL 프런티어는 해당 문서의 URL과 다운로드 기록을 캡슐화하는 객체를 저장합니다.  
  
 추상적으로 말해서, 프론티어는 클라이언트에 두 가지 주요 방법을 제공하는 URL 저장소이다. 하나는 저장소에 URL을 추가하는 방법이고 다른 하나는 저장소에서 URL을 가져오는 방법이다. 클라이언트는 URL이 추가되는 순서를 제어하는 반면, 프론티어는 URL이 반환되는 순서를 제어합니다. 즉, URL 프론티어가 크롤러의 다운로드 일정을 제어합니다.  
  
Mercator의 다른 많은 부분과 마찬가지로 URL 프론티어는 플러그형 구성 요소입니다. 이 구성 요소의 약 6개 버전을 구현했습니다. 서로 다른 버전의 주요 차이점은 예약 정책에 있습니다. 정책은 복잡성과 기어다니는 웹 서버에 제공하는 "politess"의 정도(즉, 속도 제한)에서 모두 다르다.  
  
 대부분의 크롤러는 시드 세트의 페이지에서 시작하여 웹의 너비 우선 통과를 수행하는 방식으로 작동합니다. 이러한 통과는 FIFO(선입/선입) 대기열을 사용하여 쉽게 구현된다. 그러나 웹 페이지에 상대 URL이 널리 보급되면 FIFO 대기열 내에서 호스트 인접성이 높아지며, 즉, 대기열에는 동일한 호스트 이름을 가진 URL의 실행이 포함됩니다. 크롤러의 모든 스레드가 하나의 FIFO 큐에서 URL을 큐에 대기시키면, 그들 중 많은 스레드들이 HTTP 요청을 동일한 웹 서버에 동시에 발행하여, 그것을 오버로드할 것이다. 이러한 행동은 사회적으로 용납될 수 없는 것으로 여겨진다. (사실, 그것은 일부 웹 서버를 손상시킬 가능성이 있다.)

이러한 과부하는 미결 HTTP 요청의 수를 지정된 웹 서버로 제한함으로써 방지할 수 있습니다. 이 작업을 수행하는 한 가지 방법은 지정된 시간에 특정 웹 서버에 하나의 쓰레드만 연결할 수 있도록 하는 것입니다. 우리는 이것을 약한 예의 보장이라고 부른다.

우리는 약한 politeness guarantee를 충족시키는 프론티어를 구현했고 그것을 사용하여 몇 번의 크롤링을 수행했는데, 각각의 프론티어는 수천만 개의 문서를 가져왔습니다. 각 탐색 중에 다양한 웹 서버 관리자로부터 몇 가지 불만이 접수되었습니다. 우리의 약한 politeness guarantee가 여전히 몇몇 사람들에 의해 너무 무례하게 여겨졌음이 분명해졌다. 문제는 politeness guarantee가 약해 같은 사회자에게 요청이 한 차례도 중단되지 않고 발급되는 것을 막지 못한다는 점이다.  
  
 그림 3은 가장 정교한 프런티어 구현을 보여줍니다. 지정된 호스트에 발급된 HTTP 요청의 스트림을 제한하는 더 강력한 politeness guarantee를 제공할 뿐만 아니라, 가능한 한 (공손성 요구 조건에 따라) 크롤링 스레드 간에 작업을 균등하게 분배하며, URL 다운로드 일정을 예약하기 위한 우선 순위 기반 체계를 제공한다. 프론티어는 URL 우선순위를 정하는 프런트엔드(그림의 윗부분)와 강한 공손함을 보장하는 백엔드(그림의 아랫부분)로 구성된다.  
  
 URL u가 프론티어에 추가되면 플러그형 우선 순위 구성 요소는 URL과 다운로드 내역(예: 마지막 다운로드 이후 문서가 변경되었는지 여부)을 기준으로 1에서 k 사이의 우선 순위 값 p를 계산하고 이를 프런트 엔드 FIFO 큐에 삽입합니다.

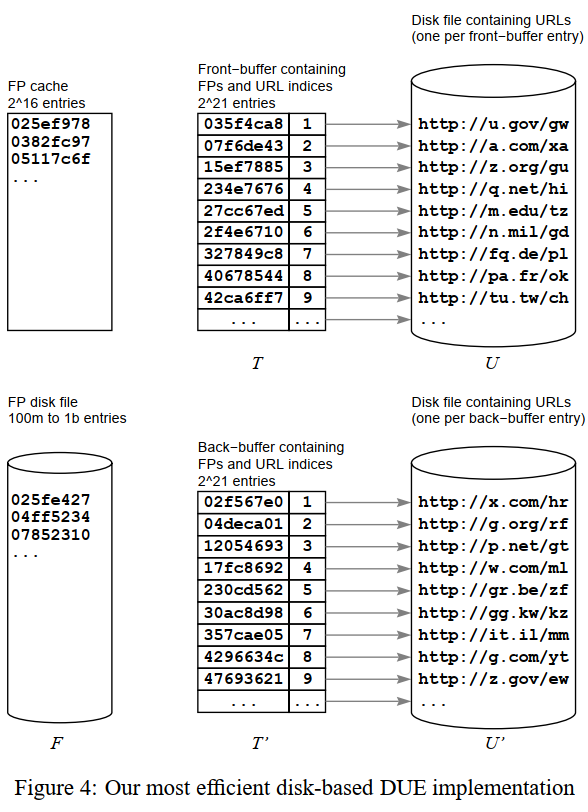


백엔드는 각각 비어 있지 않고 단일 호스트의 URL만 포함하는 FIFO 대기열과 호스트에서 백엔드 대기열로 연결되는 맵을 유지하는 테이블 T를 유지 관리합니다. 또한 각 FIFO 큐에 대한 핸들이 포함된 힙 데이터 구조를 유지하며, 큐에 해당하는 웹 서버가 다시 연결될 수 있는 시기를 나타내는 타임스탬프에 의해 인덱싱된다. 프론티어로부터 URL을 얻는 방법은 다음과 같습니다. 먼저 호출 스레드는 힙에서 루트 항목을 제거합니다(필요한 경우 타임스탬프가 과거가 될 때까지 차단). 그런 다음 해당 백엔드 대기열 q에서 헤드 URL u를 반환합니다. 호출 스레드는 이후 해당 문서를 다운로드합니다.  
  
 다운로드가 완료되면 q에서 u가 제거됩니다. q가 비어 있으면 호출 스레드가 프런트 엔드에서 q를 다시 채웁니다. 이것은 "높은 우선순위" 큐에 치우친 프런트 엔드 큐를 무작위로 선택하고 URL u’를 제거함으로써 이루어진다. 다른 백엔드 대기열 중 하나에 u’와 동일한 호스트 구성 요소를 가진 URL이 포함되어 있으면 u’가 해당 대기열에 삽입되고 q를 다시 채우는 프로세스가 진행됩니다. 그렇지 않으면 u’가 q에 추가되고 그에 따라 T가 업데이트됩니다. 또한 호출 스레드는 u의 호스트에 다시 연결할 수 있는 시간을 계산하고 해당 타임스탬프를 사용하여 q에 핸들을 다시 힙에 삽입합니다.

백엔드 대기열의 수와 속도 제한 정도는 함께 적용됩니다. 속도 제한의 정도가 클수록 기어다니는 모든 스레드를 계속 사용하려면 백엔드 대기열이 더 많이 필요합니다. 프로덕션 크롤에서는 일반적으로 크롤링 스레드보다 3배 많은 백엔드 대기열을 사용하며, 호스트에서 URL을 다운로드하는 데 걸리는 시간보다 10배 더 오래 기다린 후 해당 호스트에 다시 연락합니다. 이 값은 모든 스레드를 사용 중 상태로 유지하고 불만 사항의 속도를 최소한으로 유지하기에 충분합니다.  
  
 전체 웹에서 URL 프런티어는 곧 가장 큰 머신에서 사용 가능한 메모리보다 커집니다. 따라서 프런티어의 URL 대부분을 디스크에 저장해야 합니다. Mercator에서 각 FIFO 큐는 대부분의 URL을 디스크에 저장하고 메모리에서 머리와 꼬리 부분에 고정된 수의 URL만 버퍼링한다.  
  
3.2 효율적인 중복 URL 제거 기능

링크를 추출하는 과정에서 웹 크롤러에는 동일한 문서에 대한 여러 링크가 나타납니다. 문서를 여러 번 다운로드 및 처리하는 것을 방지하기 위해 중복 URL 제거기(DUE)는 URL 프론티어를 보호합니다. 추출된 링크는 DUE에 제출되며, DUE는 이전에 제공된 링크는 무시한 채 새로운 링크를 프론티어로 전달합니다.

DUE 구현 중 하나는 이전에 발견된 모든 URL의 메모리 내 해시 테이블을 유지합니다. 공간을 절약하기 위해 테이블에는 URL 자체보다는 URL의 8바이트 체크섬이 저장됩니다. 우리는 스펙트럼 특성이 우수하고 충돌 가능성에 대해 기하급수적으로 작은 확률적 한계를 제공하는 Rabin’s fingerprinting algorithm [3, 21]을 사용하여 체크섬을 계산한다. 체크섬의 고차 3바이트는 해시 테이블 척추에 인덱싱하는 데 사용됩니다. 오버로우 버킷의 모든 체크섬은 동일한 고차 3바이트를 가지므로 실제로는 5개의 저차 바이트만 저장합니다. 포인터 및 카운터 오버헤드를 고려하여 10억 개의 URL 체크섬을 이러한 해시 테이블에 저장하는 데는 5GB가 약간 넘는 용량이 필요합니다.  
  
 이 구현은 매우 효율적이지만 상당한 하드웨어 투자가 필요하며 메모리 요구 사항은 크롤 크기에 비례합니다. 디스크 기반 접근 방식은 이러한 문제를 방지하지만 효율적으로 구현하기는 어렵습니다. 우리의 첫 번째 디스크 기반 구현은 기본적으로 URL 지문 해시 테이블을 디스크에 저장했습니다. 널리 사용되는 지문을 메모리에 캐슁하여 DUE 제출 6건 중 1건만 디스크 액세스가 필요했습니다. 그러나 지문 인식 기능의 스펙트럼 특성으로 인해, 메모리 내 캐시에서 누락되는 지문 스트림의 인접성이 거의 없기 때문에 사실상 모든 디스크 액세스에는 디스크 검색이 필요했다. 최첨단 디스크에서는 평균 탐색에 약 8ms가 필요한데, 이를 통해 초당 125개의 검색 또는 750개의 DUE 제출을 수행할 수 있습니다. 평균 웹 페이지에는 약 10개의 링크가 포함되어 있으므로, 크롤링 속도는 초당 75개의 다운로드로 제한됩니다. 처음에는 운영 체제의 파일 버퍼 캐시에 의해 병목 현상이 발생하지만, 디스크 기반 해시 테이블이 파일 버퍼 캐시보다 커지면 검색 작업이 성능 병목 현상이 됩니다.



지금까지 설명한 두 가지 설계 모두 이전에 보지 못했던 URL을 즉시 프론티어에 추가합니다. URL 버퍼링을 통해 디스크에 대한 액세스 권한을 분할 상환할 수 있으므로 DUTION 처리량을 늘릴 수 있습니다. 그림 4는 가장 효율적인 Disk 기반 DUIL 구현의 주요 데이터 구조를 보여줍니다. 이 구현의 가장 큰 데이터 구조는 정렬된 URL 지문의 파일 F입니다.  
  
 URL u를 DUE에 제출하면 해당 지문 fp가 계산됩니다. 다음으로, fp는 인기 URL의 캐시와 메모리 내 해시 테이블 T에 대해 점검된다. 만약 fp가 어느 하나에 포함되어 있다면, 추가 조치는 필요하지 않다. 그렇지 않으면 URL 디스크 파일 U에 u가 추가되고 U에 있는 fp to u 서수의 매핑이 T에 추가됩니다.  
  
 일단 T의 크기가 미리 정의된 한계를 초과하면, "낙타의 등을 부러뜨리는" 스레드는 T의 내용을 지문/순서 쌍으로 구성된 테이블 T’ 에 복사하고, T를 비우고, U의 이름을 U′로 바꾼다. 이 매우 짧은 원자 연산이 완료되면, 다른 크롤링 스레드들은 DUE에 URL을 자유롭게 제출할 수 있고, 백브레이킹 스레드들은 각각 T’ 와 U’ 의 새로운 내용을 F와 프론티어에 추가한다. 먼저 T’를 지문 값으로 정렬한 다음 T’와 F의 선형 병합을 수행하여 T’의 모든 행에 지문이 F에 추가된 것을 표시합니다. 다음으로, T’를 순서값으로 정렬합니다. 마지막으로, T’ 와 U’ 를 순차적으로 스캔하고, T’ 로 표시된 U’의 모든 URL을 프런티어에 추가합니다.  
  
실제 크롤에서, 우리는 이 DUE 구현이 탐색 제한적인 것보다 적어도 두 배 이상 잘 수행된다는 것을 발견했다. 그러나 T’를 F로 병합하는 데 필요한 시간이 결국 T를 채우는 데 필요한 시간을 초과하기 때문에 처리량은 시간이 지남에 따라 여전히 악화된다.

3.3 DNS 문제

인터넷 상의 호스트는 32비트 번호인 IP(Internet Protocol) 주소로 식별됩니다. IP 주소가 연산기호는 아닙니다. 이 문제는 하나 이상의 IP 주소를 식별하는 cnn.com과 같은 심볼 호스트 이름을 사용하면 방지할 수 있습니다. 심볼 호스트 이름 형식으로 ID가 제공된 인터넷 사이트에 접속하는 모든 프로그램은 해당 이름을 IP 주소로 확인해야 합니다. 이 프로세스를 호스트 이름 확인이라고 하며 DNS(도메인 이름 서비스)에서 지원됩니다. DNS는 응답 내용을 찾을 때까지 이름 서버가 보다 권위 있는 이름 서버에 대한 요청을 참조하는 전역으로 분산된 서비스입니다. 따라서 단일 DNS 요청을 완료하는 데 몇 초 또는 심지어 수십 초가 걸릴 수 있습니다. 전 세계를 여러 번 왕복해야 할 수 있기 때문입니다.

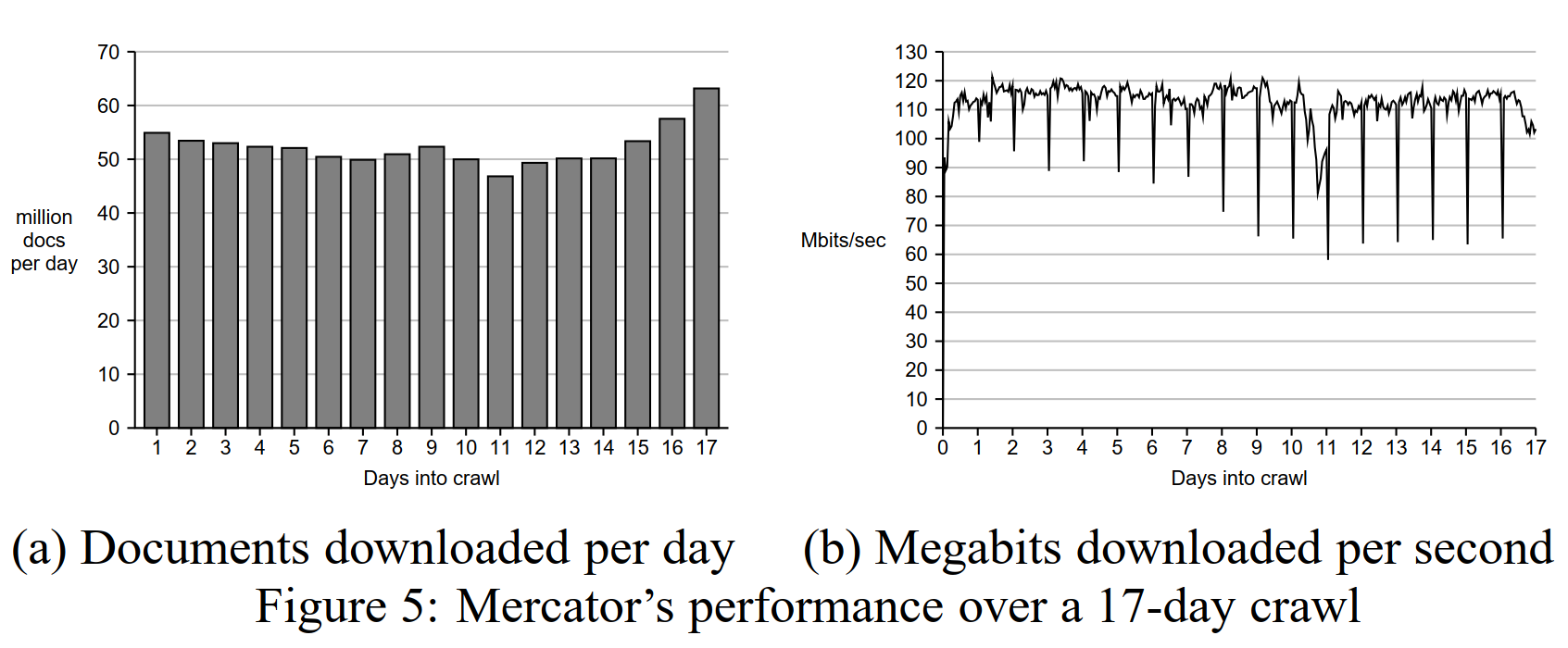
DNS 이름 resolution은 대부분의 웹 크롤러에서 잘 문서화되어 있는 병목 현상입니다. DNS 결과를 캐슁하여 이러한 병목 현상을 완화하려고 했지만 부분적으로만 효과가 있었습니다. 몇 가지 검색 후, 우리는 DNS 조회에 대한 Java 인터페이스가 동기화된다는 것을 발견했습니다. 추가 조사 결과, Unix의 대부분의 선호도에 대한 DNS 인터페이스(즉, 버클리 인터넷 이름 도메인(BIND) 배포판의 일부로 제공되는 gethostbyname function)도 동기화되어 있는 것으로 밝혀졌다. 즉, 캐시되지 않은 이름의 주소 공간당 하나의 DNS 요청만 동시에 처리될 수 있습니다. 캐시 누락률이 높아 이 제한으로 인해 심각한 병목 현상이 발생할 수 있습니다.

이러한 문제를 해결하기 위해 DNS 확인을 Mercator의 플러그형 구성 요소 중 하나로 만들었습니다. 우리는 호스트 운영 체제에서 제공하는 resolver를 사용하지 않고 로컬 네임 서버에 직접 DNS 요청을 전달하는 다중 스레드 DNS resolver 구성 요소를 구현했으며, 이 구성 요소는 각 쿼리에 대해 권한 있는 서버에 연결하는 실제 작업을 수행합니다. 여러 요청을 병렬로 할 수 있기 때문에, 우리의 해결사는 자바나 유닉스 해결사보다 훨씬 더 빠르게 호스트 이름을 해결할 수 있다.

이 변화는 상당한 크롤링이 속도 향상으로 이어졌다. 변경하기 전에 DNS 조회를 수행하는 것은 각 스레드의 경과 시간의 70%를 차지했습니다. 사용자 지정 해결기를 사용하면 경과 시간이 14%로 단축되었습니다. DNS 확인에 사용된 실제 CPU 사이클 수는 매우 적습니다. 대부분의 경과 시간은 원격 DNS 서버를 기다리는 데 사용됩니다.) 게다가, 우리의 해결사는 해상도를 병렬로 수행할 수 있기 때문에, DNS는 더 이상 병목 현상이 아니다; 만약 그렇다면, 우리는 단순히 작업자 스레드 수를 늘릴 것이다.

4 대형 크롤링의 경험

이 섹션에서는 기어다니는 실험의 결과를 설명합니다. 크롤링 클러스터는 각각 4GB의 메인 메모리, 650GB의 디스크, 100 Mbit/sec 이더넷 카드를 갖춘 4개의 Compaq DS20E 알파 서버로 구성되어 있습니다. 클러스터가 인터넷 백본에 가까이 있습니다. 우리의 ISP 속도는 우리의 대역폭을 160 Mbits/sec 제한한다.  
  
 2000년 12월, 우리는 17일[[2]](#footnote-2) 동안 8억 9100만 개의 URL을 처리한 크롤을 수행했습니다. 그림 5a는 크롤의 하루에 처리되는 URL 수를 나타냅니다. 그림 5b는 크롤의 수명 동안 대역폭 소비량을 나타냅니다. 주기적인 다운스피크는 크롤러 검사기가 하루에 한 번 상태를 가리키기 때문에 발생합니다. 크롤링은 전체 수명 동안 네트워크 제한이 있었으며 CPU 로드는 50% 미만이었으며 디스크 활동도 낮았습니다.



모든 웹 사용자가 알고 있듯이 모든 다운로드 시도가 성공적인 것은 아닙니다. 크롤링 도중 우리는 각 다운로드 시도 결과에 대한 통계를 수집했다. 그림 6은 결과 백분율을 보여줍니다. 처리된 URL 8억9100만 개 중 3500만 개는 로봇 다운로드에서 제외됐다.txt 파일과 존재하지 않는 웹 서버를 참조한 9백만 개. 즉, 크롤러는 8억 4700만 개의 HTTP 요청을 수행했다. 이러한 요청 중 7억 2,500만 건은 HTTP 상태 코드 200(즉, 성공적이었음), 9400만 건은 HTTP 상태 코드 200을 반환했으며 2800만 건은 TCP 장애를 겪었다. 인터넷에는 HTML 페이지, GIF 및 JPEG 이미지, MP3 오디오 파일, PDF 문서와 같은 다양한 종류의 콘텐츠가 있다. MIME(다목적 인터넷 메일 확장) 표준은 이러한 내용 유형에 대한 이름 지정 체계를 정의합니다 [8]. 성공적으로 다운로드된 문서의 내용 유형 분포에 대한 통계를 수집했습니다. 전체적으로 3,173개의 서로 다른 콘텐츠 유형(이 중 대부분은 일반적인 콘텐츠 유형의 오탈자)을 발견했습니다.

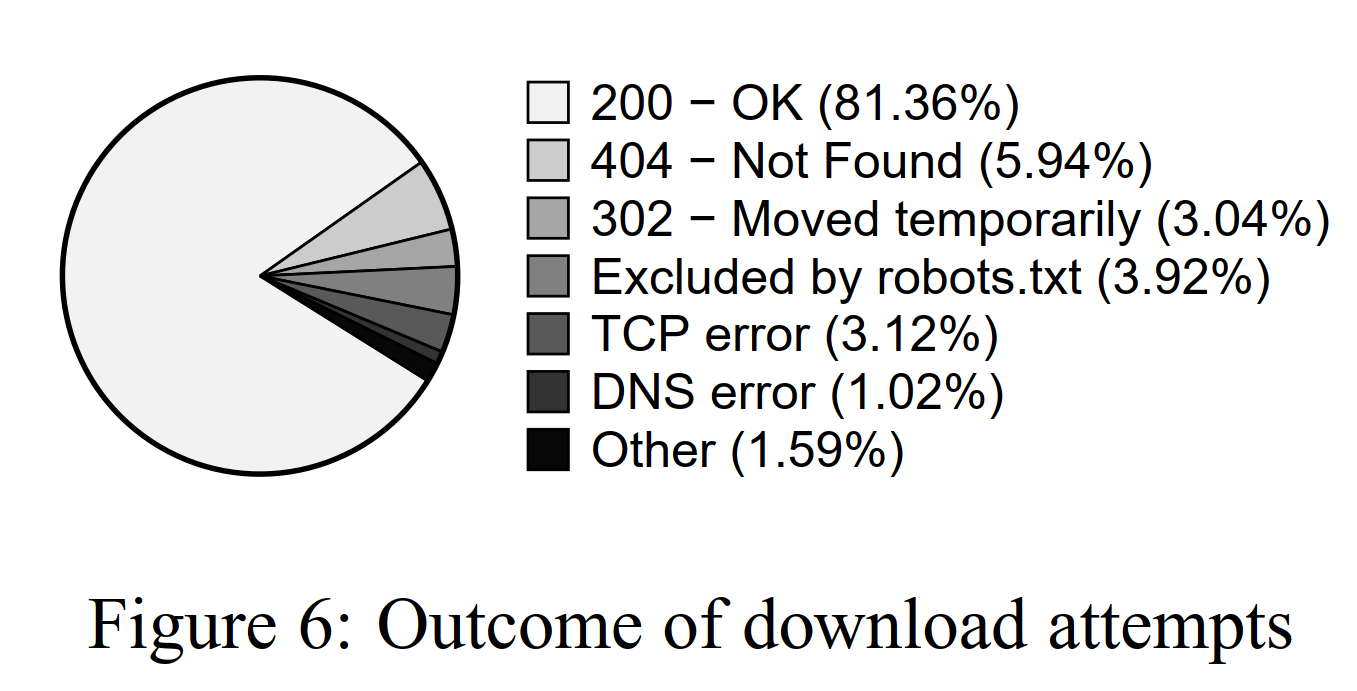


그림 7은 가장 일반적인 유형의 백분율을 보여줍니다. HTML 페이지(text/html 유형)는 모든 문서의 거의 3분의 2를 차지하며, 이미지(GIF 및 JPEG 형식 모두)는 또 다른 30%를 차지하며, 다른 모든 콘텐츠 유형을 합친 것은 5% 미만이다.

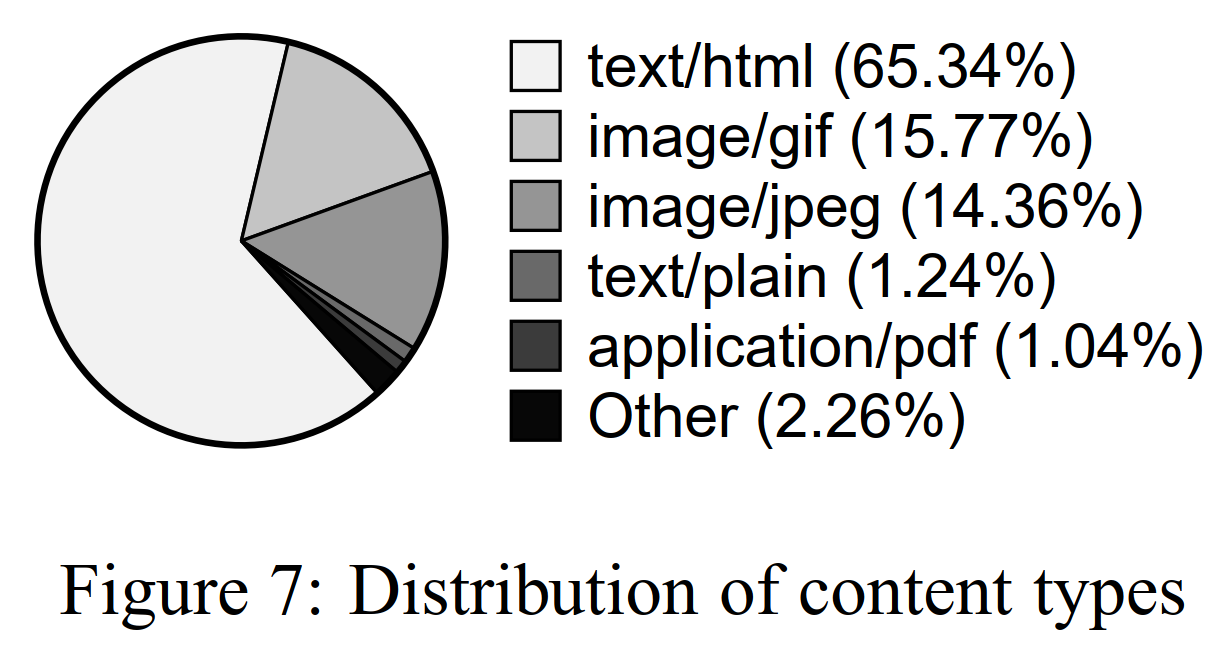


그림 8은 문서 크기 분포를 보여주는 히스토그램입니다. 이 그림에서 문서는 기하급수적으로 증가하는 문서 크기로 표시된 22개의 빈에 배포됩니다. n 크기의 문서는 n보다 크지 않은 레이블이 있는 가장 오른쪽 빈에 배치됩니다. 성공적으로 다운로드된 7억 2,500만 개의 문서 중 67%는 2K에서 32K 바이트 사이였으며, 이는 그림에서4개의 가장 큰 막대에 해당합니다.

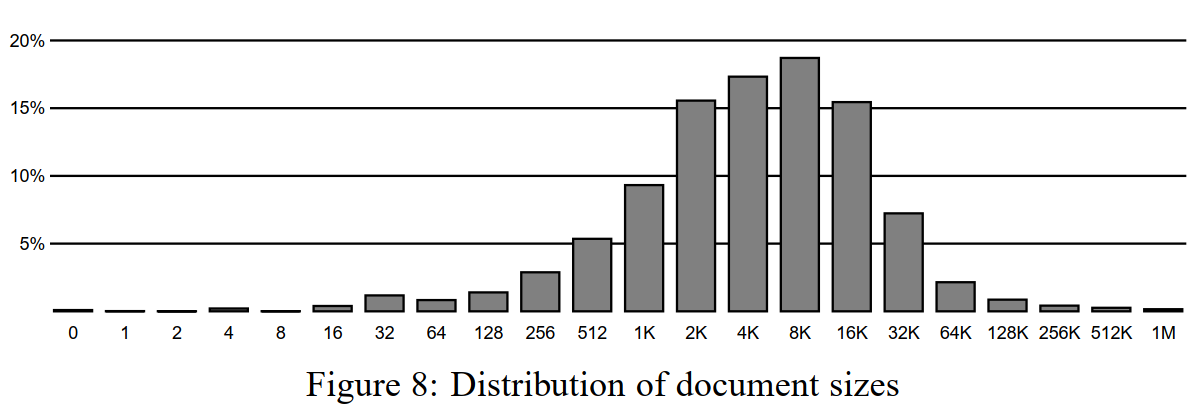
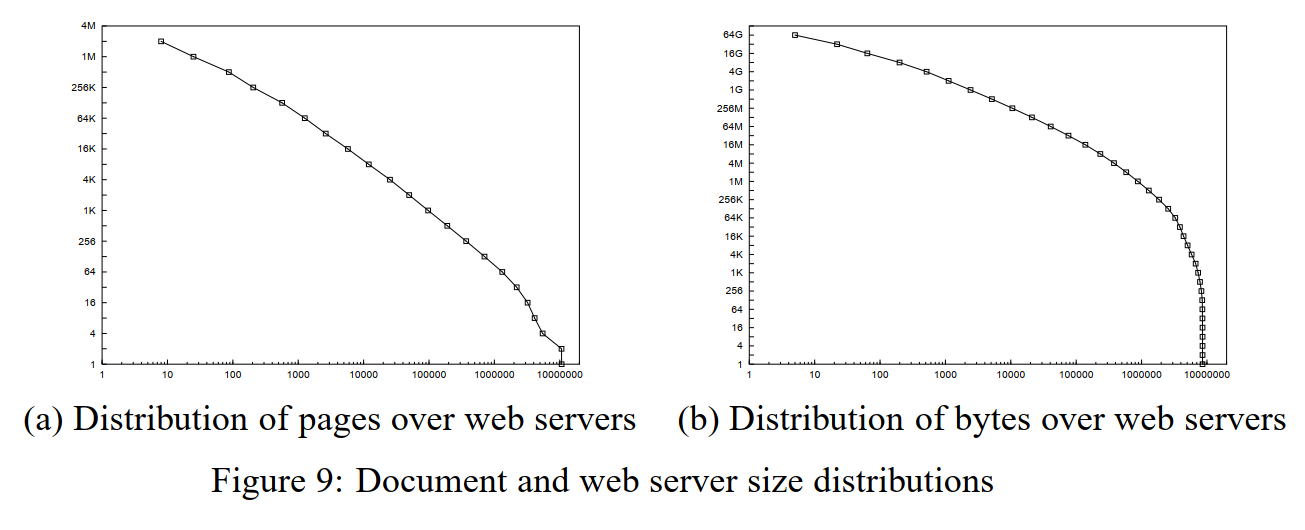
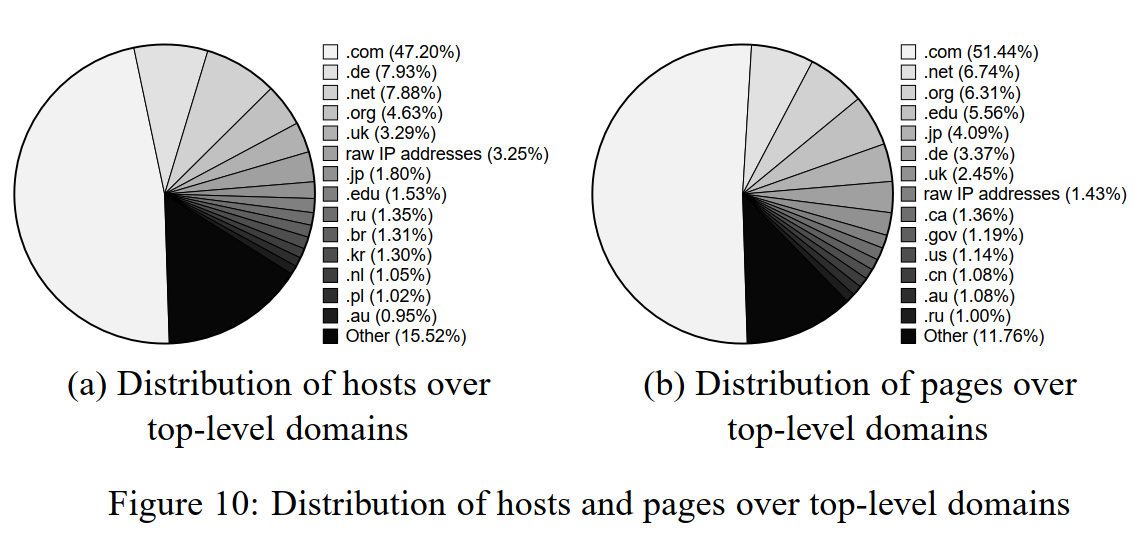


그림 9는 웹 서버 간의 콘텐츠 배포를 보여줍니다. 그림 9a는 전체 페이지의 세분성을 사용하여 내용을 측정하는 반면, 그림 9b는 바이트 단위로 내용을 측정합니다. 두 그림은 로그-로그 척도로 표시되며, 두 그림 모두에서 점(x, y)은 x 웹 서버의 페이지/바이트 수가 최소 y개임을 나타냅니다. 그림 9a의 그림에 가까운 선형 모양은 웹 서버를 통한 페이지 분포가 Zipfian임을 나타냅니다.



마지막으로, 그림 10은 최상위 도메인에 걸친 웹 서버와 웹 페이지의 분포를 보여줍니다. 서버와 페이지의 약 절반이 .com 도메인에 속합니다. 대부분의 경우 최상위 도메인의 호스트 수와 페이지 수는 서로 잘 상관되어 있습니다. 하지만, 몇 가지 재미있는 주름살들이 있습니다. 예를 들어 .edu 도메인은 호스트의 약 1.53%만 포함하지만 전체 페이지의 5.56%를 포함합니다. 즉, 대학 웹 서버의 평균 페이지 수는 웹의 평균 서버 수의 거의 4배에 달합니다.



5 결론  
  
 고성능 웹 크롤러는 많은 웹 서비스의 중요한 구성요소입니다. 고성능 크롤러를 구축하는 것은 사소하지 않은 작업입니다. 크롤러가 조작한 데이터가 너무 커서 메모리에 전혀 맞지 않기 때문에 디스크와 메모리 사용의 균형을 맞추는 방법과 관련된 성능 문제가 있습니다. 이 장에서는 크롤러에 필요한 주요 구성 요소를 열거했으며, 일부 구성 요소에 대한 설계 대안을 논의했습니다. 특히, 이 장에서는 확장 가능하고 분산적이며 성능이 뛰어난 크롤러인 Mercator를 전체적으로 Java로 기술하였다.  
  
 Mercator의 설계는 주요 크롤러 작업을 처리하기 위한 크롤러 코어와 사용자가 맞춤형 크롤러 작업을 수행하기 위해 새 모듈을 런타임에 제공할 수 있는 구성 요소 기반 아키텍처를 통한 확장성을 특징으로 한다. 이러한 확장성 기능은 매우 성공적이었다. 우리는 Mercator를 다양한 크롤링 작업에 적응시킬 수 있었고, 새로운 코드는 일반적으로 상당히 작았다(수백 줄에서 10개까지). 또한, 구성요소 모델에 의해 제공되는 afforded유연성은 동일한 기능 구성요소의 다른 구현으로 실험하도록 격려하여 새롭고 효율적인 데이터 구조를 발견할 수 있게 했다. 우리의 경험에서, 이러한 혁신은 사용자 공간 코드의 낮은 수준의 조정보다 더 큰 성능 향상을 가져왔다[13].  
  
 Mercator의 확장성 설계도 효과가 좋았다. 다양한 메모리 풋프린트에 맞게 크롤러를 쉽게 구성할 수 있습니다. 예를 들어 메모리 크기가 128MB에서 2GB인 시스템에서 실행했습니다. 다양한 하드웨어 플랫폼에 대해 Mercator를 구성할 수 있기 때문에 주어진 탐색 작업에 가장 비용 효율적인 플랫폼을 선택할 수 있다.  
  
 자바를 구현 언어로 사용한 것은 프로젝트를 시작할 때 상당한 회의론에 부딪혔지만, 우리는 그 선택을 후회하지 않았다. Java의 스레드, 가비지 컬렉션, 개체 및 예외 등 다양한 기능이 결합되어 구현이 더 쉽고 우아해졌습니다. 또한 I/O 집약적인 애플리케이션의 경우 Java는 성능에 거의 부정적인 영향을 미치지 않습니다. Compaq Alpha Servers에서 실행되는 Profile Mercator는 60% 이상의 주기가 커널과 C 라이브러리에서 사용되고 40% 미만의 JIT 컴파일 자바 바이트코드를 실행하는 데 소비된다는 것을 보여준다. 실제로 Mercator는 성능 번호가 게시된 다른 웹 크롤러보다 속도가 빠릅니다.  
  
 Mercator는 매우 인기 있는 것으로 입증되었다. 알타비스타의 서치엔진3 제품에 편입돼 알타비스타의 미국과 유럽 검색사이트 웹 크롤러로 활용되고 있다. 우리의 동료 Raymie Stata는 12 테라바이트가 넘는 웹 콘텐츠를 수집하는 Mercator 크롤을 수행했습니다. 이 크롤링은 Internet Archive의 웹 페이지 컬렉션에 기여했습니다. Raymie는 또한 2000년 미국 대통령 선거의 보도를 감시하기 위해 의회 도서관을 대표하여 연속적인 크롤링을 수행하였다. 취임 5개월 전 선거 관련 웹사이트 200여 곳의 매일 스냅 사진을 찍었다. 마지막으로, Mercator는 Compaq 내 다른 웹 연구의 원동력이 되었습니다. 예를 들어, 우리는 크롤 추적을 사용하여 주요 검색 엔진 인덱스의 품질과 크기를 추정하는 너비 우선 검색 크롤 대신 랜덤 워크를 수행하도록 Mercator를 구성했다[10, 11].

1. 이 용어는 웹 크롤러를 거미라고 부르는 일반적인 관습에서 영감을 받았습니다. 사실, 우리의 내부 이름은 Atrax입니다. Atrax는 Sydney Funnel Web Spider 라고도 알려져 있는 atrax robustus 의 이름을 따서 지어졌습니다. 이 종은 군락에 사는 몇 안 되는 거미종 중 하나입니다. [↑](#footnote-ref-1)
2. 비교해 볼 때, 현재 구글 인덱스는 약 7억 개의 완전히 색인화된 페이지를 포함하고 있다. (구글 홈 페이지에서 주장하는 인덱스 크기인 13억 5천만 개는 발견되었지만 아직 다운로드되지 않은 URL을 포함한다.) [↑](#footnote-ref-2)